FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Sensor-Based Monitoring and Management of Software Artifacts

Tiago Almeida Fernandes

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão - *Faculdade de Engenharia da Universidade do Porto*

Supervisor: Hakan Erdogmus - *Carnegie Mellon University - Silicon Valley*

July 8, 2016

Sensor-Based Monitoring and Management of Software Artifacts

Tiago Almeida Fernandes

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Ana Paiva - *Faculdade de Engenharia da Universidade do Porto*
External Examiner: João Paulo Fernandes - *Universidade da Beira Interior*
Supervisor: Rui Maranhão - *Faculdade de Engenharia da Universidade do Porto*

July 8, 2016

Abstract

Nowadays software engineers rely on continuous integration (CI) tools together with version control systems (Git, SVN, Mercurial) to build and detect eventual conflicts between concurrent changes by different developers. The tools currently being used can run test suits on the project and even run some static (e.g. code style verification) and dynamic analysis (e.g. code coverage) through additional plugins. In this scenario there is a centralised server that runs a continuous integration tool, for instance Jenkins, with a set of plugins for some extra analyses. The flow employed is such that developers make commits, the server pulls them, builds the piece of software, runs the analysis defined and notifies the team of the build outcome. This approach used to monitor and manage code artifacts is monolithic and too coarse-grained to be scalable. In addition, developers cannot register interest in specific artifacts that affect them.

CodeAware, an ecosystem inspired by sensor networks consisting of probes and actuators, aimed at improving code quality and team productivity, will be the research topic.

The ecosystem is composed of: artifacts that can go from a variable to an entire application; probes which can be static, dynamic or meta, that are agents attached to a certain artifact; controllers which are active agents that listen on probes and manage actuators; actuators which are agents that take actions like alerts, code updates, log an issue among others on behalf of a controller. To support the input of these elements in the system an IDE plugin can be used. To execute these agents is necessary an engine that sits in the CI tool.

CodeAware approach is fine-grained, distributed (each developer defines probes, controllers and actuators) and flexible/targeted (the probes are attached to artifacts).

In order to build CodeAware some challenges should be addressed such as: ensure performance; control impact on sensed artifacts; deal with code evolution; ensure scalability.

From the validation performed to the built ecosystem the goal to emphasise efficient and proactive prevention over fault localisation was understood and appreciated by the testers.

Resumo

Nos dias de hoje os Engenheiros de Software dependem em ferramentas de integração contínua (CI) juntamente com sistemas de controlo de versão (Git, SVN, Mercurial) para compilar e detetar eventuais conflitos entre mudanças concorrentes de diferentes programadores. As ferramentas a serem atualmente utilizadas podem correr *test suits* no projeto e, além disso, executar análises estáticas (e.g. verificação do estilo de código) e análises dinâmicas (e.g. cobertura do código) através de *plugins* adicionais. Neste cenário há um servidor central que corre uma ferramenta de integração contínua, por exemplo o Jenkins, com um conjunto de *plugins* para algumas análises adicionais. O fluxo empregue é tal que os programadores fazem *commits*, o servidor faz *pull* deles, compila o código, corre as análises configuradas e notifica a equipa do resultado. Esta abordagem usada para monitorizar e gerir o código é monolítica e pouco específica para ser escalável. Além disso, os programadores não conseguem especificar interesse em partes do código que os afete.

O CodeAware, que é um ecossistema baseado numa rede sensorial consistindo em *probes* e *actuators*, cujo objetivo passa por aumentar a qualidade do código e aumentar a produtividade de equipa, é o tópico de investigação.

O ecossistema é composto por: artefactos que podem ir desde variáveis até uma aplicação inteira; *probes* que podem ser estáticas, dinâmicas e *meta*, que são agentes que se anexam a artefactos; *controllers* que são agentes ativos que escutam o resultado das *probes* e gerem os *actuators*; *actuators* que são agentes que tomam ações como alertas, alterações do código, registar uma *issue* entre outros em nome do *controller*. Para suportar a inserção destes elementos no sistema pode ser usado um *plugin* para um IDE. Para executar estes agentes é preciso um motor colocado na ferramenta de CI.

A abordagem do CodeAware é específica, distribuída (cada programador define *probes*, *controllers* e *actuators*) e flexíveis/dirigidas (as probes são anexadas a artefactos).

De forma a contruir o CodeAware alguns desafios têm de ser ultrapassados nomeadamente: assegurar o desempenho, controlo do impacto nos artefactos alvo, lidar com a evolução do código, assegurar escalabilidade.

Da validação executada ao ecossistema criado o objetivo de dar enfâse a uma eficiente e proactiva prevenção sobre a localização de falhas foi compreendida e apreciada pelos utilizadores.

Acknowledgements

This master thesis was developed under two universities: Faculdade de Engenharia da Universidade do Porto and Carnegie Mellon University (CMU) - Silicon Valley. This was thanks to the Carnegie Mellon Portugal Undergraduates program to which I am very grateful. This program allowed me to stay at Carnegie Mellon University - Silicon Valley for 3 months.

I would like to thank my supervisor from Faculdade de Engenharia da Universidade do Porto – Rui Maranhão, that suggested me the CMU Portugal program and this research topic that I found very interesting. I also would like to thank him for all the support, both while I was in United States of America and in Portugal, and for the conferences that he made possible for me to attend.

I would like to thank my supervisor from Carnegie Mellon University - Silicon Valley – Hakan Erdogmus, that was the best host I could ever want. He made me feel comfortable while at CMU and his expertise and organisation were very important for the success of this project. In addition, he also presented me Mountain View, the city where the CMU campus is located, and took me to good restaurants so I could taste different cuisines. The meetings at every Friday were also very productive and fun.

I have to thank other CMU members like Diana Leathers, the ECE administrative coordinator, that every day would wish me a good morning and Uduak Eren, a master student, that was part of the CodeAware team.

I also have to thank the friends I made in Mountain View namely: Akshay Goel, Demir Yilmaz, Le Minh Tu, Praneeth Denduluri, Rajesh Ramana, Salamat Supataev, Sonica Li, Valerie Pang and Victor Dodon for making the whole experience in United States amazing letting me super motivated.

I am also very thankful to the students that were able to participate in the CodeAware validation experiment.

Lastly, I should thank my family and close friends for always supporting me and my choices.

Tiago Almeida Fernandes

"When you want to instrument a car, you don't attach sensors to the factory, but you attach them to the car."

Dr. Burak Turhan

Contents

List of Figures

LIST OF FIGURES

List of Tables

Abbreviations

CI	Continuous Integration
IDE	Integrated Development Environment
VCS	Version Control Systems
XP	Extreme Programming

Chapter 1

Introduction

The continuous battle to achieve more efficient and proactive processes in software engineering, centred in keeping engineers focused in what really matters, drives researchers to keep reinventing methodologies [Mar03]. These methodologies bring contributions to software engineering.

Continuous integration (CI) is one of those contributions that is already an intrinsic concept in the industry and open-source projects. Its core is to cyclically integrate the code base produced by different developers, build the piece of software and test it. The build outcome is later sent to developers. CI's evolution drove it to an increasing focus on monitoring software quality.

Nevertheless, there are still possibilities for improvement in software quality monitoring tools and how they integrate with CI. CodeAware is a vision based on a sensor network that goes in this direction.

1.1 Context

Behind this master thesis there is a position paper [AEP15] published by this thesis' supervisors in the International Conference on Software Engineering (ICSE) 2015, the main conference on Software Engineering worldwide. The paper presents the CodeAware vision that will be mentioned several times in this document.

This vision has the goal to renovate CI. The idea is to overcome its limitations in terms of centralisation and flexibility, namely giving developers an environment where they can have a input in the CI tool and have an output aimed specifically to them, to what they had specified. For example, a developer, in a company without code reviews, may want to know when a File is changed so he can define a *probe* that watches the file and an *actuator* that alerts him by email when the probe senses a modification (these concepts are explained later on). Note that current CI systems do not support such approach.

The inception of these new concepts together with CI inherent processes can improve individual developers view of the software they are developing, by providing a fine-grained mechanism

that can take the actions developers want in the CI server that is continuously integrating the code from the multiple developers in the project.

1.2 Problem Statement

The limitations of continuous integration fostered the CodeAware vision. The hypothesis is that with this new ecosystem both the productivity and software quality will increase by having developers focused in what is important to them. Accordingly to this hypothesis, three research questions are introduced:

RQ1 *Do developers think CodeAware empowers team productivity?*

RQ2 *Do developers think that CodeAware can help monitor and maintain code quality?*

RQ3 *Do developers find CodeAware useful?*

In order to give answers to this questions a group of developers have to try the system and then be asked some questions to understand their perception towards CodeAware.

1.3 Goals

CodeAware is a vision not having any implementation yet. The goals for this master thesis are:

- Demonstrate feasibility of CodeAware vision by implementing it.
  - Narrow down CodeAware vision to fit the thesis duration.
  - Design CodeAware architecture faithful to the vision.
  - Development of an IDE plugin in order to manage CodeAware's elements locally by the engineers.
  - Development of server engine to support CodeAware.
- Validate the usefulness of the implementation.
- Answer the research questions.

1.4 Thesis structure

Besides the Introduction, this thesis contains another 6 chapters. In chapter 2, the state of art and related works are described. In chapter 3 the CodeAware vision is detailed. Moreover, the vision is narrowed down to fit the thesis scope. In chapter 4, the CodeAware implementation is fully detailed. Technical decisions are explained as well. In chapter 5, it is explained how CodeAware can be extended, this is, add new types of probes and actuators. In chapter 6, the experiment to validate CodeAware is explained and the results presented. Finally, in chapter 7 are presented conclusions with the answers to the research questions, challenges faced and future work.

Chapter 2

Literature Review

In this chapter is presented Continuous Integration with its components and process. Tools that monitor software quality are also explored.

2.1 Continuous Integration

Continuous Integration (CI) is not a new concept in the software engineering community. Actually, it takes us back to 1998 when it was firstly introduced, nevertheless it has been evolving since then.

In 1994, Grady Booch mentioned the phrase "continuous integration" for the first time in his book "Object-Oriented Analysis and Design with Applications" [Boo94]. At the time he resorted to the phrase in order to describe the development of software using micro processes. He was already scratching the surface of what would be later introduced in 1998 by Kent Beck [Bec98].

The inception of the term "Continuous Integration" was when Beck described Extreme Programming (XP) thoroughly in his book [Bec99], after briefly introducing it one year earlier [Bec98]. XP was presented as an agile methodology intended for medium sized projects with the objective to emphasise productivity, flexibility, informality and teamwork getting programmers to achieve higher goals. This new methodology brought CI as one of its twelve practices.

Continuous integration arrived to be a problem solver, to overcome what is called "Integration Hell". Programmers used to divide the work in different modules of software. After the modules were completed they would integrate them to form the application as a whole. This process happened to be very costly both in terms of time and effort, it was very costly with plenty of bugs arising in the process. The fact that these bugs might have been introduced a long time ago and covered parts touched by different people made it even harder to detect and fix them. The idea behind CI is to minimise this problem by integrating more frequently. One can wonder why the process should be done more often if it brings such an headache to developers. In the end what happens is that if it is done frequently, like daily, then bugs are found in the same day they are introduced and the interactions between different developers' code is easily spotted. The energy and

time spent are much lower increasing the productivity. Fowler says that the effort of integration is exponentially proportional to the amount of time between integrations [FF06].

The original process described by Beck consisted of integrating as often as possible not leaving unintegrated code for more than a couple of hours. The integration consisted in merging the code, building it and then running tests. An integration that overcomes these steps was considered a successful build [Bec99].

This process may be achieved manually, but the key for the success is automating it. The process goes from a headache to a nonevent [FF06].

2.1.1 Process

Continuous Integration is a cyclic process (Figure 2.1) divided in 3 steps:

1. **Commit** — The developer changes the code, tests it and commits it to a repository.
2. **Build** — The code is fetched from the repository, integrated and built[1].
3. **Report** — Developers are notified about the build outcome.

.

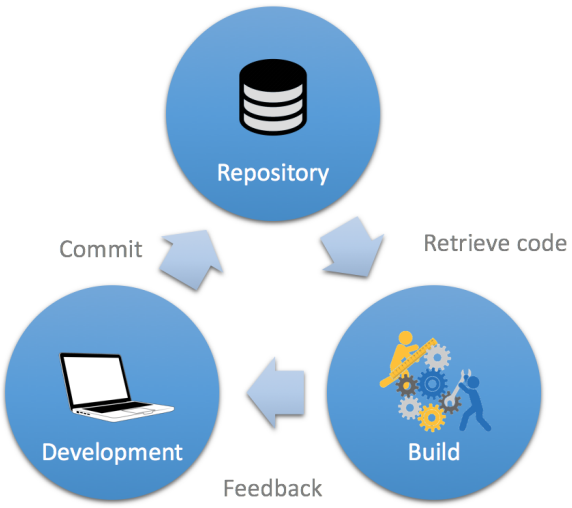

Figure 2.1: Continuous Integration cyclic flow

This cycle is dependent of the build duration. Ideally, the build is fast and the cycle runs many times a day.

2.1.2 Components

In this section components are individually introduced in order to provide more insights about the CI's process.

[1] What is called "build" includes every step performed by the CI tool.

2.1.2.1 Developers

Developers are the ones responsible for coding a piece of software. Continuous Integration philosophy [DMG07] defends some practices that they should employ:

- **Commit code frequently** — If developers commit often the amount of code changed is smaller leading to a easier process in finding bugs if they exist.
- **Do not commit broken code** — Developers should test locally the code before doing any commit. By doing this, unnecessary defects are not introduced.
- **Write automated developer tests** — Tests should be automated and have a high code coverage.
- **Fix Broken Builds Immediately** — If a build is broken every effort should focus in fixing the build. It becomes the highest priority.

By following this simple guidelines then CI empowers developers:

> "CI is the embodiment of tactics that gives us, as software developers, the ability to make changes in our code, knowing that if we break software, we'll receive immediate feedback." [DMG07, Chapter 2]

2.1.2.2 Repository

To successfully perform CI there is a central repository with the project code using the so called Source Code Management tools or Version Control Systems (VCS), where the code is accessible and developers can get the last version of the source code [DMG07, FF06]. In the old days were more common centralised version control systems like SVN[2] and CVS[3], nowadays teams have been moving to decentralised version control systems like Git[4], mostly due to lightweight branches and local and incremental commits [MBNC14]. VCS are very important for software development as they are kind of a backup for programmers. It is always possible to revert changes or go back in time, what allows experiments without the risk of breaking the working code.

The repository should contain the source code and everything else needed to do a build including: test scripts, properties files, database schema, install scripts, and third party libraries. The idea is that everything needed to run the software in a new empty machine is in the repository, excluding third party software that should be installed in the machine like Java Development Environment, Operating System or Database System [FF06].

[2] https://subversion.apache.org/
[3] http://www.nongnu.org/cvs/
[4] https://git-scm.com/

2.1.2.3 CI Server

The CI server is the engine behind Continuous Integration. Its role is to pull the latest code from the VCS, perform a build and give developers feedback of the outcome [DMG07]. There are several systems already implementing this pipeline like the open-source Jenkins[5], CruiseControl[6], Apache Continuum[7], Oracle's Hudson[8] and Bamboo from Atlassian[9].

In the CI tools early days, they used to check periodically for changes in the version control repository. If modifications were detected the server would retrieve the code, run the build scripts and then notify the team (typically by email) [DMG07]. Nowadays, platforms like GitHub[10] or Bitbucket[11], which are web-based Git repository hosting services, provide webhooks that automatically notify the integration server of modifications in the repository, preventing the constant pooling of checks for modifications.

These tools have a built in interface (Figure 2.2) to track the builds and their status, what proved to be valuable to developers. Some teams have displays always showing the status of the builds creating a *gamification* system between teams who fight for the greatest amount of successful builds [FF06].

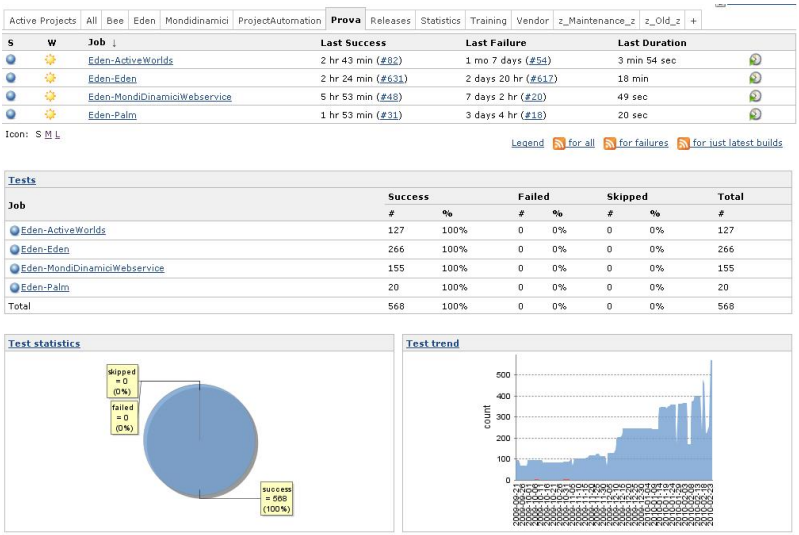

Figure 2.2: Example of a dashboard in Jenkins

Note that a CI integration server is not mandatory in Continuous Integration, the developer can always write his own scripts and manually run an integration build whenever a change is applied to the repository. However, this goes against the nature of CI about automation of processes.

[5] http://jenkins-ci.org
[6] http://cruisecontrol.sourceforge.net
[7] http://continuum.apache.org
[8] http://hudson-ci.org
[9] http://atlassian.com/software/bamboo
[10] https://github.com/
[11] https://bitbucket.org/

2.1.3 Build Pipeline

The community kept empowering the CI systems adding extra steps to the build pipeline in order to enhance the quality of the software by enforcing rules.

2.1.3.1 Automated Build

In order to have an automatic process around CI it is needed to automate the build. A lot of developers program in IDEs that already have a build system within them abstracting the programmer for the build pipeline. However, in order to run the application without the IDE, it is needed a build script or a build tool that only requires a command to have the application running.

There are plenty of tools to automate a build. For instance for Java the most common are Ant[12], Maven[13] and Gradle[14]. For Unix systems there is the old Make[15]. For Ruby there is Rake[16]. For .NET there are mainly NAnt[17] and MSBuild[18]. Most programming languages have build tools having similar behaviours among them.

With such tools the system can be built and tested within a single command making it easy to have an application up and running in a new system.

2.1.3.2 Inspections

Automated code inspections started to be part of CI including [DMG07]:

- **Static analysis** — Tools like *Findbugs*[19] which find patterns in the code that may be introducing bugs or *CheckStyle*[20] which checks for coding standards.
- **Dynamic analysis** — For example tools for test-case coverage like *EMMA*[21].

The result of such analysis is then displayed in the CI system's dashboard. These components are usually provided as *plugins*, that are developed by the community, for the CI system. Jenkins, for instance, counts with already more than 1000 *plugins*.

2.1.3.3 Deployment

With the software tested and the code inspections succeeding, the next step is deployment. CI tools also have *plugins* to support this task. Developers are currently using this automatic deployment to production step. This speeds up the process by taking advantage of the test environment that is

[12] http://ant.apache.org/
[13] https://maven.apache.org/
[14] http://gradle.org/
[15] https://www.gnu.org/software/make/
[16] http://docs.seattlerb.org/rake/
[17] http://nant.sourceforge.net/
[18] https://github.com/Microsoft/msbuild
[19] http://findbugs.sourceforge.net/
[20] http://checkstyle.sourceforge.net/
[21] http://emma.sourceforge.net/

already used anyway and reduces errors that may be introduced by manual deployments. This is usually the last step of a successful build [FF06, Mey14].

2.1.4 Value of CI

Continuous integration accordingly to Duval in [DMG07] is valuable to:

- **Reduce risks** — The fact that the system is build many times a day enables the early detection of defects.
- **Reduce repetitive processes** — Reducing repetitive processes saves time, costs and effort. Automating it turns it less error prone as the process runs the same way every time.
- **Generate deployable software** — At any time there is the last build in the server. If there is a production server and every test passes in the CI server the new version of the software can be deployed automatically to production.
- **Establish greater confidence in the software product** — The team has higher trust in the software when is constantly being informed about the build status.

2.2 CI software quality monitoring tools

One of the goals of continuous integration and its *plugins* is to monitor software quality. In order to measure it we have to check the underlying code quality. Along with the frequent releases the ambition is to keep a high quality codebase.

Over the years more and more metrics have been defined concerning the measurement of code quality [Kan02]. Furthermore, computer scientists came up with platforms to monitor software quality [GS09, MMMW05].

Towards the measurement of software quality there is a metaphor whose popularity has been growing and its principles refined [BCG+10, KNO12]. This metaphor is called Technical Debt and was introduced by Ward Cunningham when comparing poor code quality to a financial debt [Cun92].

> "In this metaphor, doing things the quick and dirty way sets us up with a technical debt, which is similar to a financial debt. Like a financial debt, the technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of the quick and dirty design choice. We can choose to continue paying the interest, or we can pay down the principal by refactoring the quick and dirty design into the better design. Although it costs to pay down the principal, we gain by reduced interest payments in the future." [Fow03]

An open-source platform that is gaining popularity is SonarQube [Son16] which has the concept of Technical Debt in its core - "Put your technical debt under control". This platform is highly customisable with *plugins* that increase the scope of the possible inspections. CI tools like Jenkins

already have *plugins* to support this platform. To measure the Technical Debt SonarQube uses the SQALE methodology [Let12a]. For this method, non functional requirements must be provided along with remediation and non remediation costs for those requirements. The remediation costs are the charge to fix the problem and are represented by the amount of time needed. The non remediation costs are represented as a ratio scale and are the value for accepting to not fix the problem and ship the software anyway, this is, the compensation for accepting violations [Let12b]. The Technical Debt is the sum of these costs and is represented in time units or amount of money [CP13].

The Technical Debt can be an accurate estimation of the software quality, refer to [Let12b] that provides more insights about the topic.

2.3 Conclusion

From this chapter some insights about Continuous Integration can be extracted. Its history and evolution are addressed as well as the process behind it. Software monitoring tools that integrate with CI systems are also covered.

The next chapters will focus in overcoming the situations when CI falls short and in software quality measured through the Technical Debt metaphor.

Chapter 3

CodeAware Vision

Nowadays Continuous Integration is very spread in the industry being used by a lot of companies. It is not difficult to start running a CI system and it brings some advantages to the developers, that avoid repetitive processes, and to the company, that can have a global view of the project in the CI system dashboard [DMG07]. Nevertheless, CI has margin to grow.

Despite all the extensibility through *plugins* the systems can be limiting in terms of flexibility. The build pipeline is achieved through centralised configuration of the continuous integration tool and its *plugins*. Engineers cannot register interest only in code artifacts that affect them, what starts to make sense when the codebase is large [AEP15].

Changing the paradigm of a centralised server containing the build configurations of the whole project, to a perspective where the configuration can be attached to the artifacts may achieve a granular, and individually definable, controllable, and actionable behaviour that cannot work otherwise. Developers can focus in what they are interested.

In this chapter this new vision will be described.

3.1 Motivation

A sensor network has multiple sensor nodes that are small and lightweight, and actuators. The outcome of the sensors are signals based on sensed effects that are then transmitted. The actuators take actions based on the signal received by the sensors [Sta08]. The intuition is that perhaps the codebase can be treated like a collection of artifacts to which we can attach sensors and actuators like a sensor network achieving a distributed approach instead of a centralised configuration in the CI server.

3.2 Vision

CodeAware vision describes an ecosystem based on sensor networks aimed at improving code quality and team productivity. In this vision the artifacts are the objects that need to be probed, they can go from variables to entire classes.

The ecosystem (Figure 3.1) is composed of soft agents that integrate the codebase:

- **Probes / Sensors** — Are attached to the artifacts. They can be detectors that perform static analysis to find common faulty problems in the code (e.g. FindBugs and CheckStyle) or dynamic analysis for instance to measure the code coverage by the tests. They can also be meta-probes by aggregating input of multiple probes providing higher level information. The result of their job is a signal that can be caught by controllers.

- **Controllers** — Controllers may subscribe the input from multiple probes and then take actions according to what they listen. Filters can be specified in order to refine the probes output of interest.

- **Actuators** — Controllers' strategies are deployed into actuators who actually perform the desired action. Actuators can be of different types providing different kinds of behaviours:

  - *Notifier* – Notifies interested subscribers (developers or service) of an event of their concern.
  - *Recommender* – Makes recommendations to interested subscribers based on related actions.
  - *Updater* – Modifies an artifact.
  - *Reporter* – Reports an event to a target external system (e.g. issue tracking systems) or generates a static analysis report for the engineers to review.

This approach changes the paradigm of having a centralised configuration in the CI server to a distributed one. The configuration stays together with the codebase and attached to the artifacts. This turns the configuration fine-grained, individually definable and controllable by each developer.

The hypothesis of this vision is:

> "Our hypothesis is that the CodeAware ecosystem will improve both productivity and software quality by bringing relevant changes, not only external ones caused by updates in dependencies but also internal changes within the codebase, to the attention of the software engineer before the fact in a manageable and targeted way, thus emphasizing efficient and proactive prevention over fault localization and fixing." [AEP15, Section I]

A comparison between current CI tools and CodeAware can be spotted in Table 3.1.

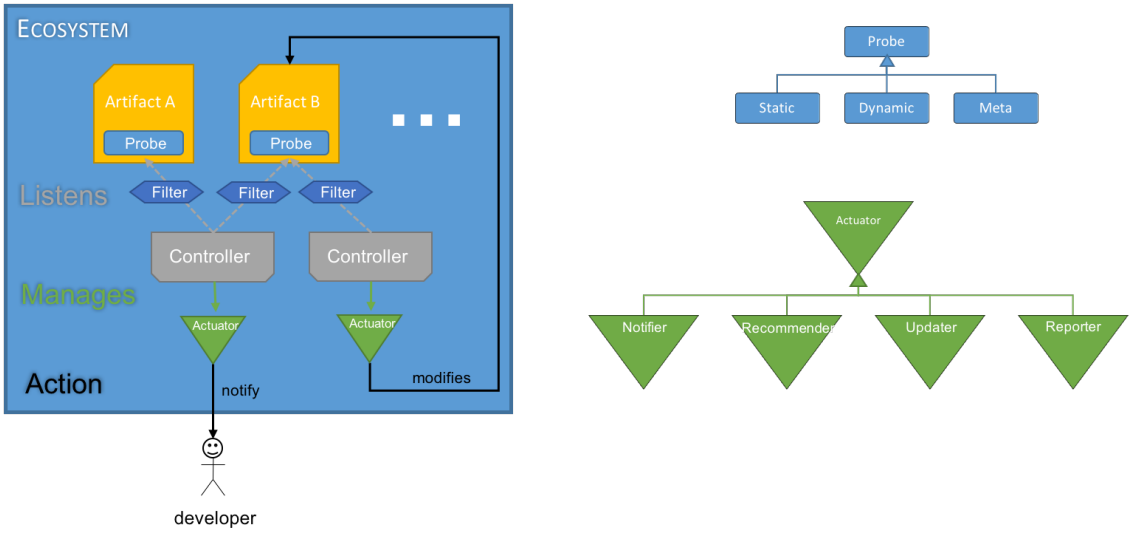

Figure 3.1: CodeAware ecosystem

3.3 Example Scenario

Sue is a game developer with the role to ensure that the game her team is developing runs at 60 frames per second so that players can have an enjoyable experience. This requirement implies that the `render` method, that draws each frame to the display, should not take more than 16 milliseconds. Sue decides to create a CodeAware probe, through her CodeAware IDE plugin, that notifies her every time a performance decrease is detected:

1. Creates a *probe* that profiles the artifact every time a new version of the `render` method is committed to the repository. The probe consists of running a test suit multiple times, the resulting signal will be equal to the maximum execution time of the artifact.

2. Defines an *actuator* of type notifier that can send her an email.

3. Creates a *controller* that listens to the *probe's* signal. If the value carried in the signal is greater than the 16ms then the controller actuates the email notifier.

On Sue's next commit, these agents will be saved in the repository along with any other changes she had made. CodeAware will detect the definition of these agents and in the next commits it will bring to Sue's attention if her game performance dropped. This example scenario was adapted from [AEP15].

3.4 Architecture

The CodeAware system is twofold. There is the Client Side Engine which lives inside the IDE as a plugin and the Server Side Engine that sits in the CI tool's build pipeline (Figure 3.2).

Table 3.1: Comparison between current CI tools and CodeAware

	CI tools	**CodeAware**
Configuration	centralised	distributed
Configuration location	CI server	repository
Configuration access level	dev-ops	developer
Automated Build	✓	✓
Tests Support	✓	✓
Code Inspections	✓	✓
Developer specific inspections		✓
Notifications	✓	✓
Developer specific Notifications		✓
Automatic Deployment	✓	✓
IDE integration	✓	✓
Developer personalised actions		✓

3.4.1 Client Side Engine

The Client Side Engine as previously mentioned is integrated in a *plugin* for an IDE. The goal is that the developer does not have to leave his development environment to create a CodeAware probe keeping his focus in what he is doing. The plugin consists of a panel where probes and actuators can be configured. In addition, it has the capability to easily attach probes to artifacts.

3.4.2 Server Side Engine

The Server Side Engine is encapsulated in the CI tool's build pipeline by adding extra steps to it. CodeAware's server engine pipeline can be observed in Figure 3.2 and is described in the following steps:

1. The *Dispatcher* separates probes, actuators and controllers.

2. The *Optimizer* optimises the probes so that their signals are generated efficiently.

3. The *Signal Generator* verifies if the running conditions of the probes are satisfied and then the corresponding analysis are performed, the output is kept in a signal table.

4. The *Orchestrator* is fed with the controllers by the *Dispatcher* in order to serialise them and resolve any eventual dependencies.

5. The *Processor* receives the actuators from the *Dispatcher* and executes the serialised controllers by getting the input from the signal table. Then executes the actuators defined in the controller.

6. The effects specified are achieved.

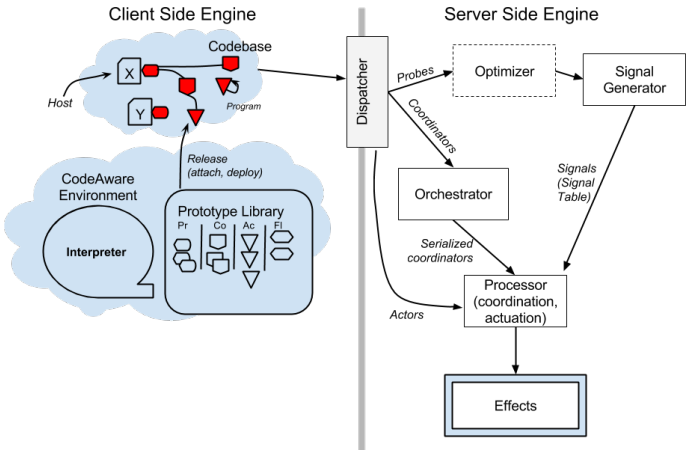

Figure 3.2: CodeAware Architecture

3.5 CodeAware - Thesis Scope

CodeAware is a complex system and poses a number of research challenges touching multiple research fields [AEP15]. Thus, in order to fit this master thesis, CodeAware vision had to be narrowed down. In this section, it is described what was cut out from the original CodeAware description.

3.5.1 Ecosystem

From the original ecosystem the changes are:

- *Filters* were removed.
- The *Controllers* became a container that encapsulates an *artifact*, a *probe* and an *actuator*.
- *Meta-probes* were removed. Initially static and dynamic probes are enough to validate the system.
- Only *Actuators* of type *Notifier* and *Reporter* are kept. Nevertheless, development was in a direction that the others can be easily included later.
- In an initial stage, developers will not be able to create their own probes and actuators, they will only be able to customise the ones provided.

3.5.2 Architecture

The architecture needs some minor adjustments to cope with the changes in the ecosystem, namely the change in the *controllers* concept.

The Client Side Engine remains the same but now has to encapsulate the *artifacts*, *probes* and *actuators* in *controllers*.

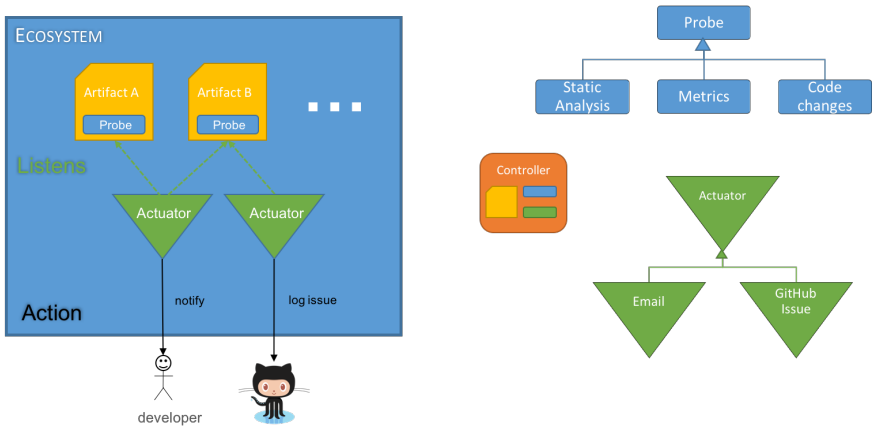

Figure 3.3: CodeAware ecosystem simplified

In the Server Side Engine the pipeline was changed along with the development as new challenges kept rising. Nevertheless, the idea was always to keep the concepts the most similar possible with the original architecture: the *Orchestrator* role changes to cope with *actuators* with the exact same behaviour in between them. An *Host Resolution* component was added in order to verify if the *artifacts* to which *probes* and *actuators* are attached still exist. The flow was also changed to a more parallel approach. In the Chapter 4 the CodeAware prototype architecture will be fully detailed.

3.6 Conclusion

In this chapter was presented an overview of the CodeAware vision. As it is very broad, its underlying concepts were narrowed down to fit in the scope of this Master Thesis. From this chapter on, when referring to CodeAware and its components, it is referring to the simplified version.

Chapter 4

CodeAware Implementation

In this chapter the CodeAware implementation, that is a contribution from this thesis, will be fully detailed along with explanations of the decisions made. It will start with a overview of all the modules and how they interact with each other and then it will dive into each of the modules specific implementation details.

4.1 Modules

CodeAware implementation was divided in three separate modules. There are two that were expected according to CodeAware specification and another one added because there were common dependencies between the other two modules.

- **Client Side Engine - IntelliJ IDEA[1] Plugin** — The client side engine that integrates with an IDE.
- **Server Side Engine - Jenkins[2] Plugin** — The server side engine that sits in the build pipeline of a CI tool.
- **CodeAware Common** — The module that has common parts between CodeAware's client and server sides.

4.2 CodeAware Common

This module contains components that are shared between the client and server side. It was created because it would not make sense to have separate implementations of the same thing in different parts of CodeAware. This would lead to inconsistencies and is not a good practice. Code duplication is actually classified as a code smell [VEM02].

[1] https://www.jetbrains.com/idea/
[2] https://jenkins.io/

The common parts are related with the probes, actuators, controllers and artifacts definitions and properties. All these components are shared between the server and client side. In the client, they are defined and then the server takes actions based on their configurations. Another common part is the `ContentManager` that is responsible for saving and reading the CodeAware generated data and configurations from the repository.

4.2.1 Implementation Design

The CodeAware Common implementation consists of five packages (Figure 4.1). The *disk* package contains what involves reading and writing from disk. The components package contains the previously described CodeAware components divided in packages as well. Note that an *host* is an artifact to which a probe was attached.

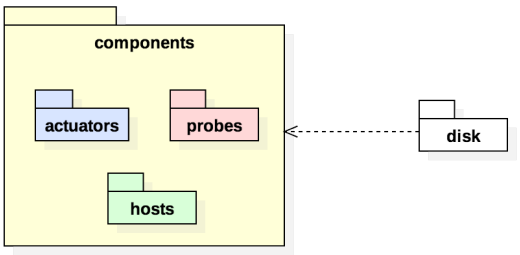

Figure 4.1: CodeAware Common packages diagram

In Figure 4.2 the classes diagram can be spotted (note that classes' colours match the package, in Figure 4.1, to what they belong). Some classes need further clarification:

- **CodeAwareProbe** — The base abstract class any CodeAware prove must extend. The minimum requirement is a name for the probe type so it can be presented to the final user in a human friendly way. The probes' classes can then contain whatever other configuration details they may need.

  The probes implemented are *FindBugs* (static analysis), *CheckStyle* (metrics analysis) and *DiffChecker* (check for changes in the code).

- **CodeAwareActuator** — The base abstract class any CodeAware actuator must extend. As well as with the probes, the minimum requirement is a name. However, actuators can and should implement two functions:

  - `divide` — An Actuator should know how to divide itself in more fine-grained units. This method returns an array of *CodeAwareActuator*s. For instance if an Email actuator can hold multiple emails, the divide method should return multiple *CodeAwareActuator*s each one with one email only.
  - `group` — This method receives a *CodeAwareActuator*, if it is possible to group that actuator with the receiving one it groups it and returns true, otherwise returns false.

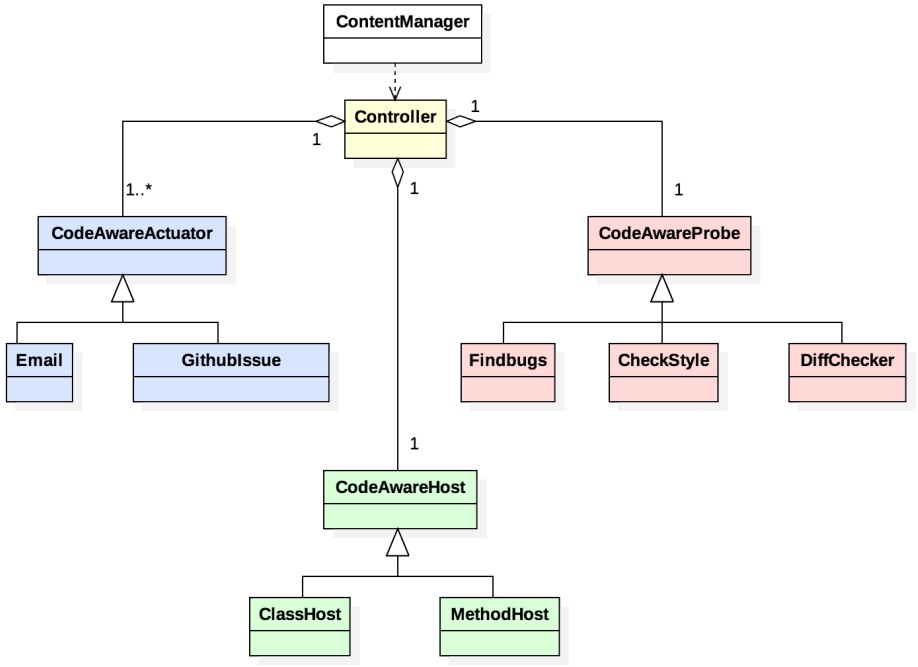

Figure 4.2: CodeAware Common classes diagram

Back to the Email example, if one instance of Email (A) receives another instance of Email (B), instance A adds instance's B emails to its list.

The importance of these functions is later addressed when describing the Orchestrator component in CodeAware server side in section 4.4.1.

The actuators implemented were *Email* that sends emails to the email addresses provided by the user and *GithubIssue* that creates an issue in GitHub[3] Issue Tracking System.

- **CodeAwareHost** — The base abstract class any CodeAware host type must extend. It contains the host path, name and qualified name (for classes it is the classes with packages, for example: *pt.up.fe.codeaware.Hello.java*).

  In order to deal with code changes that may change an host, it is important to have a way to compare the similarity between hosts. For this reason, a method `getProximityPoints` that receives as argument another *CodeAwareHost* returns a float representing the distance between the two hosts.

  For the prototype the two subclasses implemented were *ClassHost* and *MethodHost* that respectively represent a class or a method.

  The heuristic used for the `getProximityPoints` for the two kinds of hosts is pretty simple but it also proved to be quite effective. Firstly, if the hosts are of different type it returns -1. When they are of the same type it is based on the class or method's signature. The

[3] https://www.github.com/

signature was divided in the categories presented in Table 4.1 (note that not all categories are present in *ClassHost* and *MethodHost*). All the categories except the name are binary, this is, or they are equal between hosts obtaining the points or are different getting no points. The name category is a linear value between 0 and 5 based on the distance between the two names. The distance algorithm used was Levenshtein Distance [4].

Table 4.1: Categories used for similarity points calculation between hosts

Category	**Points**	***ClassHost***	***MethodHost***
Name	5	✓	✓
Modifiers	1	✓	✓
Return type	2		✓
Parameters	2		✓
Implemented interfaces	1	✓	
Extended class	1	✓	

- **Controller** — The Controller is the class that is able to encapsulate a probe, actuators and a host. It contains a label so that the user can recognise it and a creation time.

- **ContentManager** — Provides an *API* to export and import CodeAware data to the repository. It exports / imports the data to / from a *JSON* file. The data referred are the Controllers. The Controllers encapsulate a probe, actuators and the artifact (or host) to which the probe is attached. This *JSON* file is a representation of the Controller that is saved in the repository along with the code in a folder called ".codeaware". Each *JSON* file has a unique name based on the host's file name and on the time it was created.

4.2.2 External dependencies

This module has some external dependencies that should be remarked:

- **Gson** [5] — Is a library that allows the conversion of Java objects to a JSON representation. It can also do the other way around. It is an open source library supported by Google.

  It is used by the *ContentManager* to export / import the controllers to / from the disk.

- **Reflections** [6] — Eases the use of reflection to find subclasses of a class. It works by scanning the classpath and indexing the metadata.

  It is used in multiple places in order to obtain all subclasses of *CodeAwareProbe*, *CodeAwareActuator* and *CodeAwareHost*. It is used to assure that when a new component is developed it can be added to CodeAware without any change to CodeAware base code.

[4] https://en.wikipedia.org/wiki/Levenshtein_distance
[5] https://github.com/google/gson
[6] https://github.com/ronmamo/reflections

- **Cloning** [7] — It is a library that can create a deep clone of any Java object. It recursively clones everything under an object.

  It is used to clone CodeAware components, mainly controllers. It makes it possible for developers when adding new components to CodeAware not needing to implement the clone method of their component.

4.3 Client Side Engine - IntelliJ IDEA Plugin

IntelliJ IDEA was the chosen IDE to extend. The reasons are related with personal preference and to a increasing adoption by students and industry. The other possible choice would be Eclipse [8]. The cost of choosing IntelliJ IDEA over Eclipse was a much slower development because there is a lot less documentation online. A large part of the development process was checking out other plugins' source code to learn how certain things could be achieved. In the end it worked out very well.

4.3.1 Implementation Design

The client side has a more complex architecture in comparison with the common part. In Figure 4.3 the packages are represented. Note that the common package represents the CodeAware Common module. The packages organization is as follows:

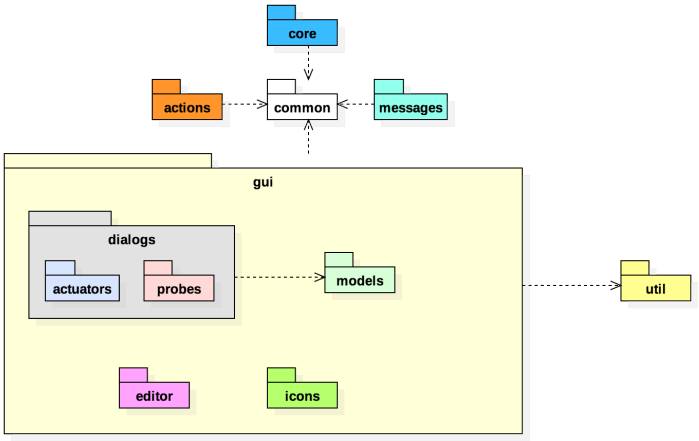

Figure 4.3: CodeAware IntelliJ IDEA packages diagram

- **gui** — Contains everything related with user interface.
  - **dialogs** — Contains the dialogs for probes and actuators configurations.
  - **models** — Contains custom models to feed tables and list boxes in the CodeAware IDE's main panel.

[7] https://github.com/kostaskougios/cloning
[8] https://eclipse.org/

  - **icons** — Manages icons references for usage in the context menus.
  - **editor** — Deals with presentation of icons and options dialogs where probes are attached to in the code.

- **util** — Contains some functions to facilitate the interaction with IntelliJ plugins API.
- **actions** — Contains the context menu possible actions.
- **core** — Contains hooks for IntelliJ events like loading or closing a project.
- **messages** — Messages needed to have a communication between the context menu actions and the CodeAware main panel.

In Figure 4.4 this module's classes diagram can be observed (note that classes' colour match their parent package colour). From the packages description the classes behaviour are easily deducted. Nevertheless, there are some that need further clarification:

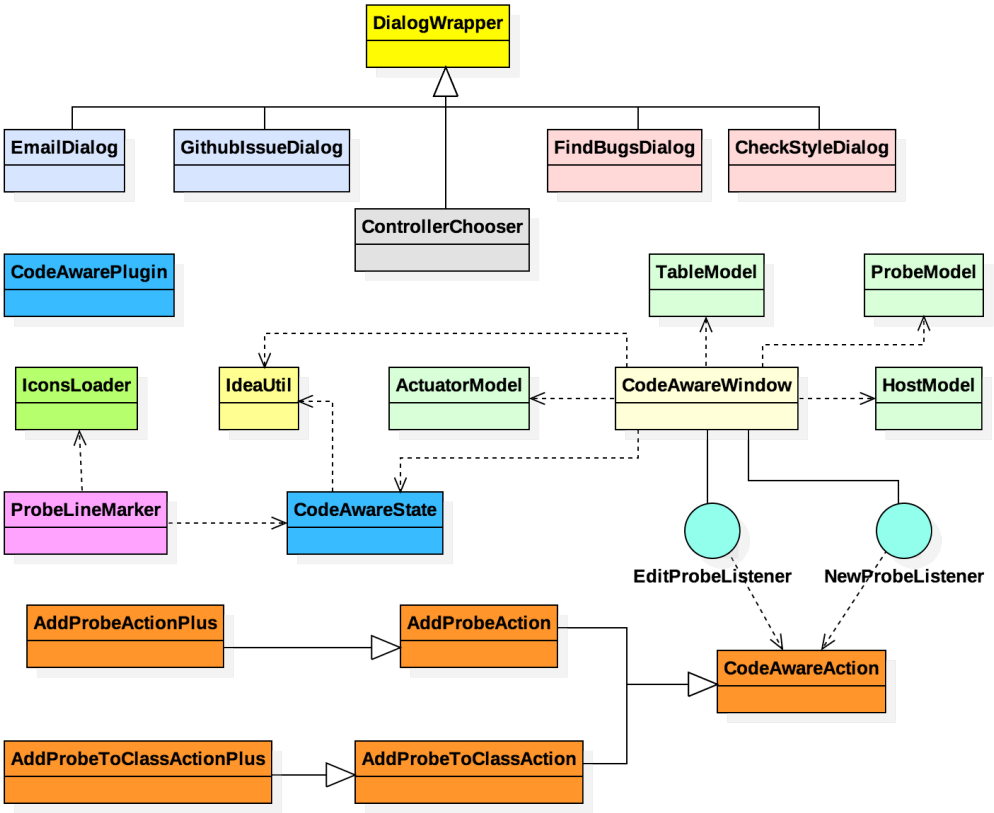

Figure 4.4: CodeAware IntelliJ IDEA classes diagram

- **DialogWrapper** — It is part of the IntelliJ plugins API. Every probe or actuator that has a configuration has to extend it. The name of the subclass follows a convention. It adds to the actuator or probe class name the key word "Dialog". This convention must be followed so

that the CodeAware IntelliJ plugin can know what dialog should be displayed when configuring certain CodeAware elements.

- **ControllerChooser** — It is a window that presents in a tree view all the controllers in the project divided by host file (Figure 4.8). The key usage for this dialog is for choosing a controller to be cloned. The intuition is that a developer may want to copy a probe and / or an actuator and attach it to another host.

- **CodeAwareState** — It is a bridge between what is saved in disk and what is kept in memory. The controllers are saved in disk but are also kept in memory because IntelliJ is constantly running background processes for context menus and editor line markers (icons on the left side of the editor). Continuously loading the controllers from the disk was slowing down the IDE.

- **CodeAwareAction** — Actions that are triggered when user interacts with context menus (Figure 4.7). The subclasses that end with "Plus" include the choice of a controller to be cloned.

- **CodeAwareWindow** — It is the main CodeAware panel in the IDE (Figure 4.6 bottom). It is always listening for messages resultant of *CodeAwareAction*s in order to keep the panel updated.

4.3.2 User Interface

The Use Case diagram in Figure 4.5 represents the actions that can be taken by the user while using the CodeAware IntelliJ plugin.

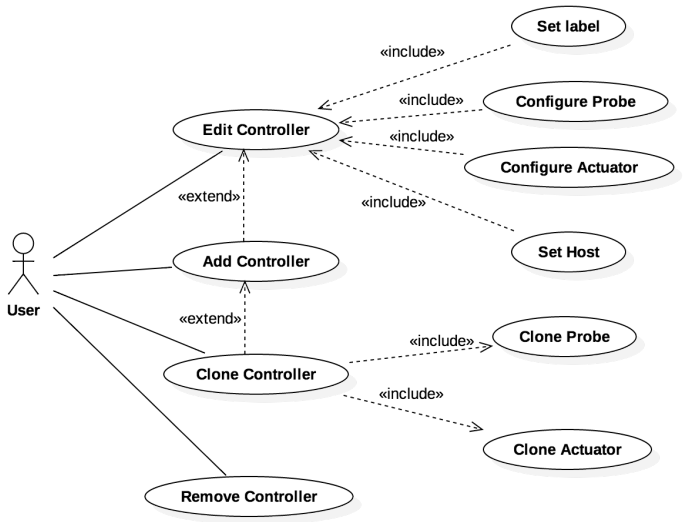

Figure 4.5: CodeAware IntelliJ IDEA Use Case diagram

The core of the plugin sits in a panel in the bottom part of the IDE that can be hidden or shown when desired. Figure 4.6 shows the panel when visible. On the left side of the panel are listed the controllers that have probes attached to the file open in the editor. On the right side it is possible do define a label for the controller, choose the host, the probe type and actuator type and then it is possible to configure the probe and the actuator by clicking in the "Config Probe" or "Config Actuator" buttons respectively. If you click the button "..." you can clone the desired component (probe, actuator or a whole controller) from another controller.

In the left side of the editor next to the function `removeAndKillWorker` an icon can be noticed. This icon means that there is a probe attached to that method. By clicking the icon the user can quickly delete or edit the selected controller (this is: the probe, the actuator, the host or the whole controller).

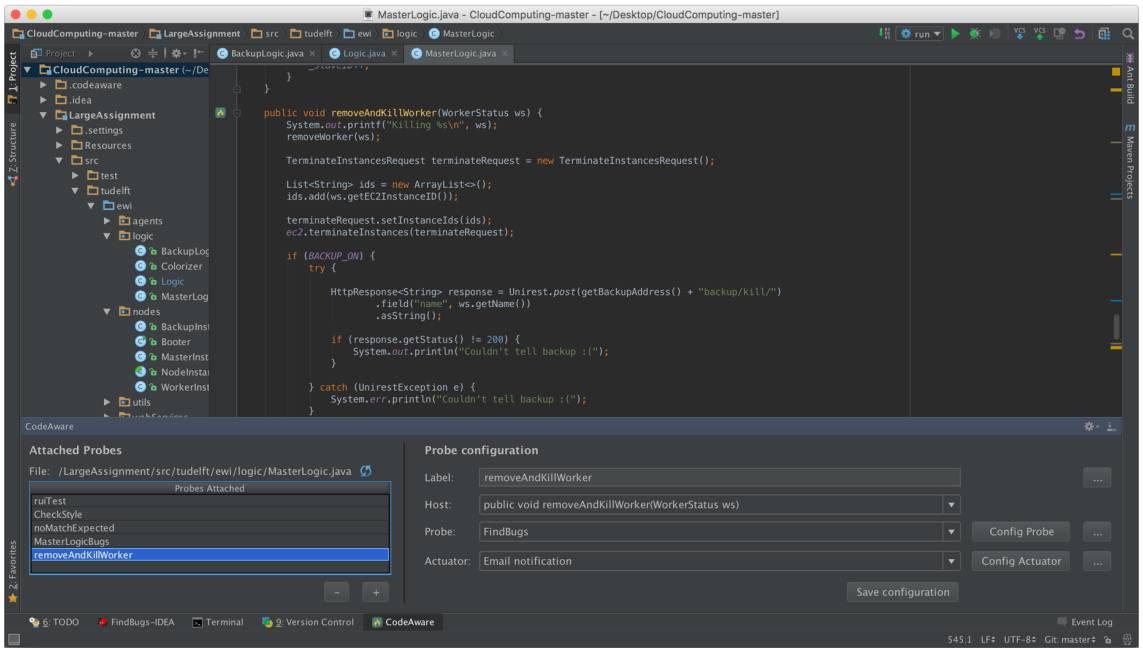

Figure 4.6: CodeAware IntelliJ IDEA plugin

Probes can be attached through the plugin main panel or through the context menu. In Figure 4.7 there is the result of right clicking in the `removeUselessWorkers` method. The CodeAware context menu provides the possibility to add a probe to the method, to clone a controller and attach it to the selected method or do the same but for the whole class. It is also possible to create and attach probes directly in the context menu from the project view (left panel in Figure 4.6 that shows the project directories and files).

The dialog presented in Figure 4.8 is a tree view of the controllers attached to files (Java classes) in the project. It was decided that this would be a good view to pick from the existing controllers one that a user may want to clone and apply in other artifact.

When a developer makes a change to the code that leaves a probe orphan, this is, the host to which it was attached disappeared, a popup is shown notifying the user. If the user clicks in the popup the CodeAware main panel will show up and suggest the user what it thinks is the new host

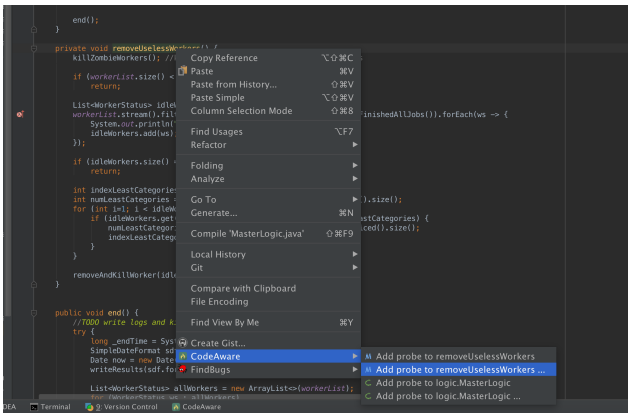

Figure 4.7: CodeAware IntelliJ IDEA plugin context menu

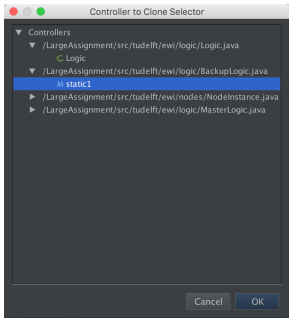

Figure 4.8: CodeAware IntelliJ IDEA plugin clone dialog

for the probe. Figure 4.9 shows this. The method `addSlave` was renamed to `addWorker`. In the top right corner the popup can be seen. By clicking the link in it the CodeAware main panel is shown and is accurately suggesting that the new host is the `addWorker` method. All the developer should do now is hit the "Save" link in the panel's balloon.

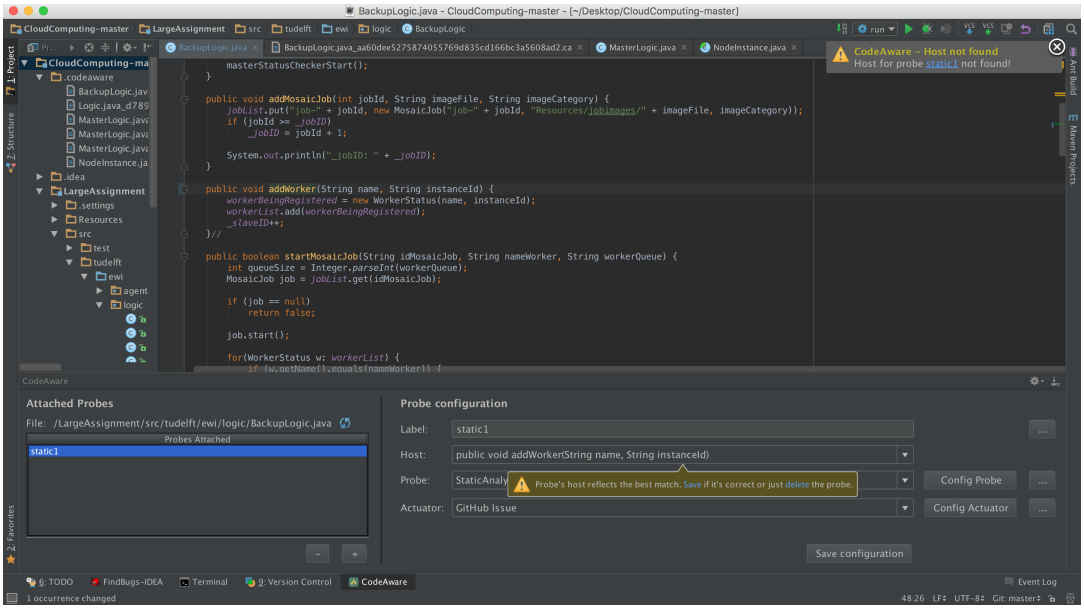

Figure 4.9: CodeAware IntelliJ IDEA plugin orphan probe warning

4.4 Server Side Engine - Jenkins Plugin

Jenkins was chosen as the Continuous Integration tool because it is the leading open-source automation server. In addition, it is very customisable containing more than 1000 plugins [Jen16]. Notwithstanding, developing a plugin is not as simple due to lack of good documentation. In order to understand how plugins really work it is imperial to go look other plugins source code.

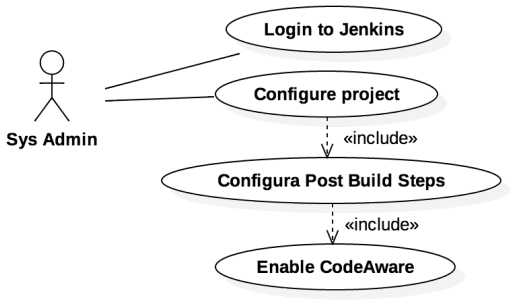

Figure 4.10: CodeAware Jenkins plugin activation Use Case diagram

The CodeAware Jenkins Plugin was developed as a post-build plugin, what means that the plugin is run in a project, if enabled, after each time the project is built. The process of enabling CodeAware Jenkins plugin in a project is described in the Use Case diagram in the Figure 4.10.

4.4.1 Implementation Design

In order to go further and explain the detailed implementation of CodeAware's engine it is important to understand its flow. For that refer to Figure 4.11. The components behaviour was changed during the development:

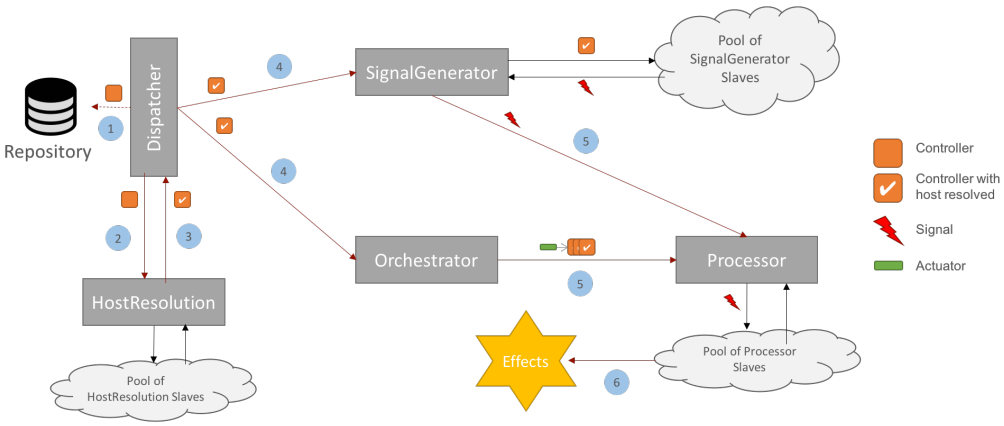

Figure 4.11: CodeAware Jenkins plugin workflow

- **Dispatcher** — The *Dispatcher* is responsible for loading the controllers from the repository and then dispatch them to the next components.

- **HostResolution** — The *HostResolution* has the important role of checking whether the host to which a probe is attached exists, this is, check if the probe is not orphan. If the probe is orphan but it can find an approximate match all the process will be done for the approximate host. In the final results there will be a notice expressing that an approximate host was used. If there is no matching host nor an approximate match the probe is dropped.

So that multiple host resolutions can be done parallelly there is a pool of *HostResolution* slaves. The number of slaves is fixed. It can be specified and should be adequate to the CPU architecture in order to benefit from parallelisation. The tasks are assigned by the *HostResolution* using the round robin scheduling algorithm [9].

- **SignalGenerator** — Just like *HostResolution*, *SignalGenerator* has a pool of slaves to which it assigns the tasks of running the probes specified in the controllers. For instance, among others, they perform FindBugs static analysis to the host specified in the controller.

  Each probe in each controller emits a signal after the probe execution. If the probe does not trigger an actuator, in FindBugs case if there is not any bug in the host, the signal generated is empty.

- **Orchestrator** — The *Orchestrator* role is to group controllers by actuators that do the exact same thing. It returns a map from actuator to a set of controllers that may trigger that actuator. For instance, if there is an actuator that sends an email to a developer A, and this developer has multiple probes with the same actuator, this is, send an email to A. It is better that A receives only one email with all the results grouped instead of being spammed with multiple emails. In order to achieve this, it is essential that *CodeAwareActuator*s' methods `group` and `divide`, described in section 4.2.1, are implemented. The example provided is quite straightforward, but it is possible to imagine different scenarios such as email actuators that contain a subset of common emails, it that case *Orchestrator*, extracts the common subset and creates a new actuator with the subset and creates a grouping from that new actuator to the controllers that trigger them.

- **Processor** — The *Processor*, just like *HostResolution* and *SignalGenerator*, has a pool of slaves in charge of taking parallel actions on its behalf. The slaves receive a set of signals and then they trigger the actuator with the information received in the signals. The desired effects happen after each slave run.

  It is important to note that the *Processor* holds the signals in a signal table and only sends them to a slave when it is already in possession of all the signals for a certain actuator (the mapping from actuator to controllers that comes from the *Orchestrator* plays a role here).

The work flow starts when the *Dispatcher* loads the controllers from the repository. Then it sends them to the *HostResolution* that returns the controllers with the host resolved. Afterwards, the *Dispatcher* sends the controllers to the *SignalGenerator* and the *Orchestrator*. They start doing their tasks parallelly. Eventually, the *Orchestrator* finishes grouping the controllers by actuator and sends the mapping to the *Processor*. At any time before or after receiving the mapping from the *Orchestrator*, the *Processor* starts receiving the signals from the *SignalGenerator*. When the *Processor* has all the signals (a signal per controller in the mapping from actuator to controllers) for an actuator it runs the actuator through a slave. The desired effects are achieved.

[9] https://en.wikipedia.org/wiki/Round-robin_scheduling

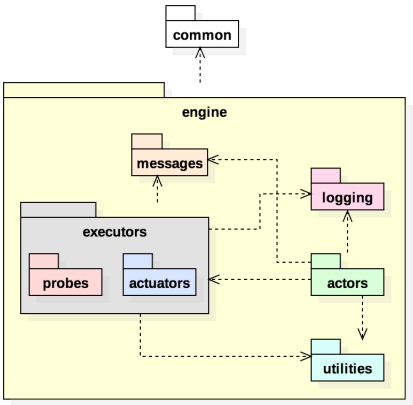

Figure 4.12: CodeAware Jenkins plugin packages diagram

The packages diagram is shown in Figure 4.12. The architecture's objective was to achieve a clean and easy to understand design. The packages content is the following:

- **engine** — Everything in the CodeAware Jenkins plugin is inside this package. It depends on the Common module that is represented as a package.
- **executors** — Contains the implementation of what was called executors. Executor is the Strategy in the Strategy Design Pattern[10] and the implementations of the Executors for the probes and actuators are the concrete strategies. The name of the executors follows a convention, they have the same name as the probe or Actuator but end in "Executor". For example, the probe "FindBugs" has an executor "FindBugsExecutor".
- **logging** — Contains mechanisms to deal with the logging in the plugin.
- **utilities** — Contains helper classes.
- **actors** — The components presented in Figure 4.11 are represented as actors and are present in this package.
- **messages** — Contains the types of messages that actors use to communicate between each other.

In the Figure 4.13 the classes diagram is presented. There are a lot of dependencies represented, most of them are between actors and messages symbolising that an actor uses messages of that type. Most of the classes need further clarification:

- **CodeAwareBuilder** — It is the Jenkins plugin entry point. It deploys the *Dispatcher* actor that deploys the other actors and starts the workflow. A *BuildInformation* object is also created and passed to the *Dispatcher*, containing information about the environment provided by Jenkins that can be useful for the other actors and executors.

[10] https://en.wikipedia.org/wiki/Strategy_pattern

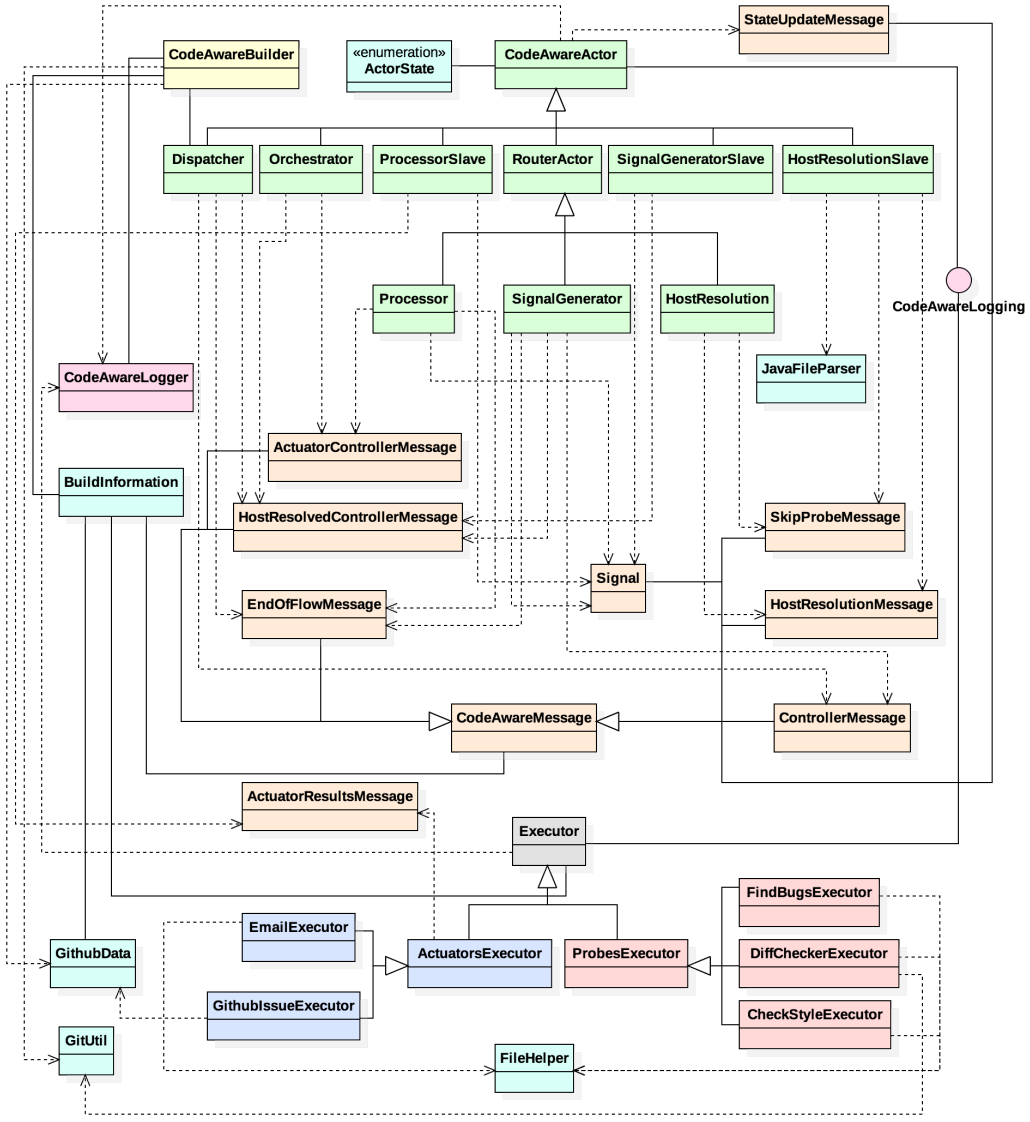

Figure 4.13: CodeAware Jenkins plugin classes diagram

- **BuildInformation** — Contains environment information like build number, workspace path among other useful information.

- **RouterActor** — It is an actor that can have a pool of slaves. It is called route because it can route tasks to its slaves.

- **CodeAwareLogger** — Stores the logging stream for each individual Jenkins' project. A choice was made in the sense that the logger should be accessible through a class instead of being an object always being passed as an argument. In order to allow multiple projects to run at the same time there must be a mapping from Jenkins project to logger.

- **CodeAwareLogging** — An interface to force the actors and executors output to be properly formatted.
- **JavaFileParser** — Parses a Java file, it is useful to look for the hosts in the Host resolution process.
- **CodeAwareMessage** — Different kinds of messages where needed to send different kinds of information. Having multiple types of messages made it easier to filter messages as soon as they are received instead of having to check how they are composed and then take action.
  - **ControllerMessage** — Message containing a controller sent from the *Dispatcher* to the *HostResolution*.
  - **HostResolutionMessage** — A message sent from the *HostResolution* to a *HostResolutionSlave*. It encapsulates multiple controllers whose hosts are in the same file. If this grouping was not done and each controller was sent individually to a *HostResolutionSlave*, Java files containing multiple hosts would be parsed as many times as the number of hosts. This way the file is parsed once.
  - **HostResolvedControllerMessage** — Message containing a controller with a resolved host.
  - **EndOfFlowMessage** — If multiple messages are being sent from one actor to another sometimes is useful to tell that all messages have already been sent.
  - **SkipProbeMessage** — This message is sent from *HostResolutionSlave*s to the *HostResolution* when there is no match nor an approximate match for the host.
  - **ActuatorControllerMessage** — This kind of message is sent from the *Orchestrator* to the *Processor* containing the mapping from actuator to controllers.
  - **Signal** — It contains the result of running a probe. Signals can contain a message and/or the path to a file. The *ActuatorExecutor*s will then receive signals and will know how to handle that information. The *EmailExecutor* for instance sends in the email's body the text message and attached the file in the signal.
  - **ActuatorResultsMessage** — A message that contains an actuator and the signals that were triggered for that actuator. It is sent from the *Processor* to a *ProcessorSlave* when *Processor* already has all signals for a certain actuator. In the end, it is an encapsulator of multiple signals.
  - **StateUpdateMessage** — Each time an actor's state is changed it notifies its parent. For all actors the parent is the *Dispatcher* except the slaves whose parent is a *RouterActor*. When the Dispatcher has the state of all actors as "Finished" it knows that everything was finished by all actors.
- **ActorState** — It is an Enum useful to know when the CodeAware workflow finished. Each actor has a state and changes it according to what they are doing.

- **GitUtil** — It provides methods to add tags to the git repository (in each build a tag is added to the repository), allows to checkout different versions. It also provides a method to do a diff between files.

- **Executor** — It is the base class of all the executors. Every Executor must implement the method "execute" where the logic happens. For an *ActuatorsExecutor* the "execute" method is called by a *ProcessorSlave* and for a *ProbesExecutor* it is called by a *SignalGeneratorSlave*.

4.4.2 External dependencies

Some external libraries were used to fasten the development, and because are known for good performance:

- **Akka** [11] — The description provided by the authors is: "Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.". Simplifying, it is an actors library. All the parallellism employed in the architecture is thanks to Akka that manages it. The pools of actors are also used without any pain. Akka was a good choice. If it was not used probably most of the process would be sequential rather than parallel.

- **JavaParser** [12] — A Java parsing library. It parses Java files. Was used mostly for finding the hosts in the host resolution step.

- **JGit** [13] — A Java library to run git commands.

- **mylin.github** [14] — A library to easily connect to GitHub. It was used by the *GithubIssueExecutor* to post issues to GitHub.

4.5 Conclusion

In this chapter the CodeAware implementation was fully detailed. It is divided in three modules that together form the CodeAware Ecosystem. The system was designed to be easy to understand and easily extensible. In the next chapter it is addressed how to extend CodeAware by adding new Probes and Actuators.

[11] http://akka.io/
[12] https://github.com/javaparser/javaparser
[13] https://eclipse.org/jgit/download/
[14] https://github.com/eclipse/egit-github/tree/master/org.eclipse.mylyn.github.core

Chapter 5

CodeAware Extension

CodeAware was developed to be easily extendable. Thus, in this chapter it will be described how simple it is to add new Probe and Actuator's types to CodeAware. As mentioned in Chapter 4, CodeAware is divided in three modules. The Common Module and the Jenkins Plugin module require an action while the IntelliJ Plugin module can or cannot be touched. All the steps of the process will be addressed.

5.1 Add a new type of Probe

In this section it will be explained how a new probe type can be added to the ecosystem. The process is divided in the three CodeAware modules.

5.1.1 CodeAware Common

As it was said in section 4.2.1, probes must extend the *CodeAwareProbe*. The Listing 5.1, shows the implementation of a new probe *MyNewProbe*.

```
1 package taf.codeaware.common.components.probes;
2
3 public class MyNewProbe extends CodeAwareProbe {
4     public final static String TEXT = "MyNewProbe";
5
6     public MyNewProbe() {
7         super(TEXT);
8     }
9 }
```

Listing 5.1: Example implementation of a new Probe

The *CodeAwareProbe* constructor receives the name of the probe type. If the probe has fields, there should be a constructor with all that fields. This is because of the Gson library used for exporting / importing the objects to / from a JSON representation.

This example is pretty simple, it can be completely customised as far as it is extending the *CodeAwareProbe*, calling its constructor with the probe name and having a constructor with all non transient fields.

5.1.2 IntelliJ IDEA Plugin

This part is only needed if it is desired that the user can insert probe's configurations through a dialog. If developed, when the user presses the "Config Probe" button, in the CodeAware main panel, the dialog is displayed. For instance, in the FindBugs probe implementation, user is presented with a dialog where the types of bugs to analyse can be selected by the user.

```
1  package taf.codeaware.gui.dialogs.probes;
2
3  import com.intellij.openapi.ui.DialogWrapper;
4  import taf.codeaware.common.components.probes.MyNewProbe;
5
6  public class MyNewProbeDialog extends DialogWrapper {
7      private MyNewProbe probe;
8
9      public MyNewProbeDialog(Project project, MyNewProbe probe) {
10         super(project, false);
11         this.probe = probe;
12
13         setTitle("MyNewProbe Configuration");
14         setModal(true);
15         init();
16     }
17
18     @Override
19     protected void doOKAction() {
20         super.doOKAction();
21         // Do whatever needed when user presses ok button
22     }
23 }
```

Listing 5.2: Example implementation of a dialog for a new Probe

In Listing 5.2 is presented the base code. Note that there is a naming convention. It has the same name as the probe with "Dialog" appended.

5.1.3 Jenkins Plugin

In this module stays what happens in the Continuous Integration server. In case of the FindBugs probe it was running the FindBugs static analysis.

The parameters of the constructor must be the ones presented in Listing 5.3. The objects will be built through reflection and these are the arguments that will be provided. In the *execute* method there is access to the *buildInformation*, the *controller*, the *probe* and the *host*.

```
1  package taf.codeaware.engine.executors.probes;
2
3  import taf.codeaware.common.components.Controller;
4  import taf.codeaware.common.components.probes.MyNewProbe;
5  import taf.codeaware.engine.messages.Signal;
6  import taf.codeaware.engine.utilities.BuildInformation;
7
8  public class MyNewProbeExecutor extends ProbesExecutor {
9      public static final String NAME = MyNewProbeExecutor.class.getSimpleName();
10
11     public MyNewProbeExecutor(Controller controller, CodeAwareHost approximateHost,
           BuildInformation buildInformation) {
12         super(controller, approximateHost, buildInformation);
13     }
14
15     @Override
16     public Signal execute() {
17         // Do what the probe is supposed to do
18
19         return new Signal.Builder(controller, true /* trigger actuator? */,
20           "This is the signal result text", host /* ProbesExecutor member */)
21                 .resultFile("path to result file") // optional
22                 .build();
23     }
24 }
```

Listing 5.3: Example implementation of a new Probe executor

There is also a naming convention here. To the name of the probe it is appended “Executor”.

5.2 Add a new type of Actuator

In this section it is explained how to create new actuators to add to CodeAware. There are some similarities with the adding new probes process.

5.2.1 CodeAware Common

Every Actuator should extend the *CodeAwareActuator* class. Just like in the probes' case the super class constructor should be called with the actuator name, and in case there are fields there must be a constructor containing all of them. In the Listing 5.4 there is the base for a new actuator. The section 4.2.1 explains the methods that should be implemented.

```
1  package taf.codeaware.common.components.actuators;
2
3  public class MyNewActuator extends CodeAwareActuator {
4      public final static String TEXT = "MyNewActuator";
5      String foo;
6
7      public MyNewActuator() {
8          super(TEXT);
9      }
10
11     public MyNewActuator(String foo) { // constructors with the field
12         super(TEXT);
13         this.foo = foo;
14     }
15
16     @Override
17     public boolean equals(Object obj) {
18         // should return true if the actuators have the same behaviour
19     }
20
21     @Override
22     public int hashCode() {
23         // Implement a proper hash function
24         return 0;
25     }
26
27     @Override
28     public CodeAwareActuator[] divide() {
29         CodeAwareActuator[] actuators;
30         // separate the actuator in multiple actuators
31         return actuators;
32     }
33
34     @Override
35     public boolean group(CodeAwareActuator actuator) {
36         if(actuator instanceof MyNewActuator) {
37             // do what is needed to join the actuators
38             return true;
39         }
40         return false;
41     }
```

```
42 }
```

Listing 5.4: Example implementation of a new Actuator

5.2.2 IntelliJ IDEA Plugin

If the actuator needs a configuration dialog then it should be added just like in the example for the probe in section 5.1.2. The only difference is the constructor's second parameter that should be of type *MyNewActuator* instead.

5.2.3 Jenkins Plugin

The actuators are the CodeAware common components that take effects that the user want performed. Like an email that sends a probe's report. In this module it is where this behaviour is specified. In the Listing 5.5 is presented a boilerplate for the implementation.

```
1  package taf.codeaware.engine.executors.actuators;
2
3  import taf.codeaware.common.components.actuators.MyNewActuator;
4  import taf.codeaware.engine.messages.ActuatorResultsMessage;
5  import taf.codeaware.engine.messages.CodeAwareMessage;
6  import taf.codeaware.engine.utilities.BuildInformation;
7
8  public class MyNewActuatorExecutor extends ActuatorsExecutor {
9      public static final String NAME = MyNewActuatorExecutor.class.getSimpleName();
10
11     public EmailExecutor(ActuatorResultsMessage message, BuildInformation
           buildInformation) {
12         super(message, buildInformation);
13
14         probesNames = message.getProbesString();
15     }
16
17     @Override
18     public CodeAwareMessage execute() {
19         // the effects wanted
20     }
21 }
```

Listing 5.5: Example implementation of a new Actuator executor

Remember once again that there is a naming convention. To the actuator class name must be appended "Executor".

5.3 Conclusion

From the code snippets presented it is clear that extending CodeAware by adding additional probes and actuators is very straightforward. There is no need to change the CodeAware platform code. Only what is being added has to be programmed following some simple guidelines presented in this chapter.

Chapter 6

CodeAware Validation

In order to understand if the CodeAware prototype that was built has potential, a simulated experience was performed. The main goal was to understand if developers when using CodeAware understand its benefits.

Some students from the last year of Mestrado Integrado em Engenharia Informática e Computação at Faculdade de Engenharia da Universidade do Porto where invited to take part in the experiment. Seven people were picked to participate.

6.1 Users' background

To know the users, a pre-test was made that asked some questions about their developer skills and team-work experience.

6.1.1 Developer Skills

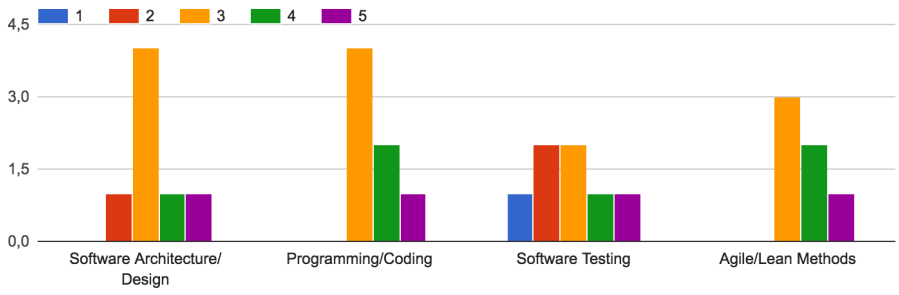

Figure 6.1: Users' knowledge as developers

The users are all programming for more than 3 years with the majority programming for 5 years. Their knowledge as developers was divided in four categories and is summarised in Figure 6.1. As CodeAware for now is limited to Java it was asked their knowledge about it. From

the seven, one said his knowledge was “beginner, occasional use” while all the others said it was “working knowledge, regular use”. The familiarity with IntelliJ IDEA is more broad and is presented in Figure 6.2. Git knowledge goes from “working knowledge, regular use” to “expert” with only two people reporting “beginner, occasional use”.

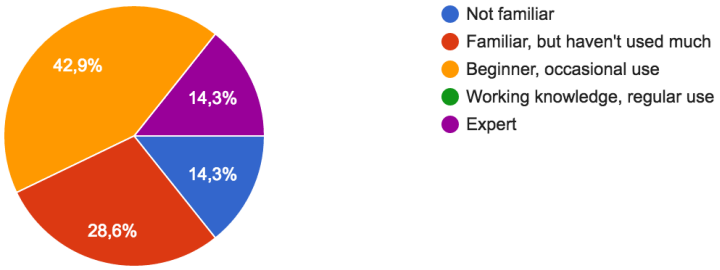

Figure 6.2: Users’ familiarity with IntelliJ IDEA

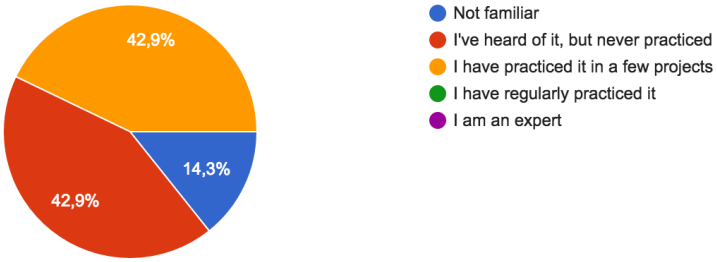

Figure 6.3: Users’ familiarity with Continuous Integration

When it comes to continuous integration the scenario changes (Figure 6.3) with only three of the users having already practised it in a few projects. Nevertheless, only one of them did not know Jenkins.

From the answers it was clear that the users had enough developer skills to judge CodeAware. They are aware of the the software engineering parts that CodeAware touches.

6.1.2 Team work

CodeAware makes sense in a collaborative environment so it was important to find out how the selected users behave (Figure 6.4). Most of them participate in software development in a team “often”.

In terms of how users deal with code changes, all of them have already dealt with frequent communication to cope with them and committing code very often. Four of them already experienced unwanted changes when dealing with merge conflicts and two used systematically pull requests. All of the users use Git, four of them use it “frequently” while three use it “sometimes”.

The users were asked to present shortcomings of the current VCS like Git and the common point was that these tools are not that easy to learn. About CI they referred as shortcomings the

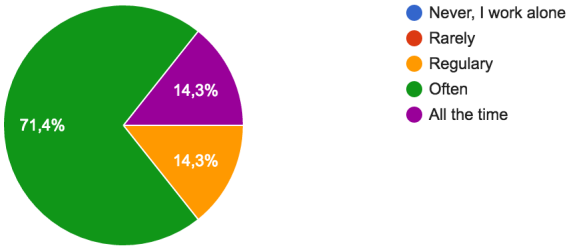

Figure 6.4: Users' software development in team frequency

fact that sometimes the automatic build fails and it is not very easy to fix and another point was that usually it is difficult to use it because the "business guys" can not understand the benefits.

The enquired users are team players and have already experienced code changes problems due to working in a team. They understand VCS and have a overview of CI.

6.2 Setup

The experiment was designed for three people. The users where divided in two groups of three people each. One additional user teamed up with another one having one team with four elements but two were pair programming.

Users were given a virtual machine with the environment ready. Intellij IDEA was installed and preloaded with the CodeAware Intellij IDEA plugin. Users were added to a GitHub repository that was forked from the *jpacman-framework*[1] repository that is used by Arie van Deursen, professor from Delft University of Technology, to teach software testing. The code was modified to remove some features that the users had to implement.

The experiment was divided in three tasks. Each task was different for each individual in a group. The groups were doing the same tasks but in different branches of the repository, so, it is only described the tasks of a group because the other group did exactly the same thing. The goal was only for the users to realise what CodeAware could do for them. Users could only pass to the next task after everyone finishing it and committing the code or probes to the repository. Between tasks a build in the Continuous Integration server was performed to let the users see what happens there.

For the first task users were presented with a narrative that asked them to add probes to specific parts of the code mentioning that they were responsible for that part of the code or saying that they wanted to pay attention to a certain component. Each user added two probes: a metrics or static analysis probe and a diff probe (code changes). They were asked to choose an email actuator. After the first task a build was performed in the CI server and no probes were triggered.

In the second task users were provided with a narrative that asked them to input code in some places to add some features. Some code provided was buggy (but not easily detectable) and other

[1] https://github.com/SERG-Delft/jpacman-framework

was just correct. The point is that users were touching each others code, where probes were attached, without knowing. After everyone finished, a build was performed in the CI server and this time probes were triggered and users received emails notifying them that some of the probes they installed were triggered. Users realised that they were touching each others code and that some of them were introducing bugs. The bugs were fixed by the owner of the artifact with the help of the report provided in the email.

The task three was exactly like the second task. It was to reinforce what was happening and in what CodeAware was helping them.

6.3 Results

After the users performed the tasks, whose goal was to understand what CodeAware could do for them, a post-test was done and the results are presented in this section. The post-test was divided in four sections.

6.3.1 Tasks

All the users completed the three tasks and classified them in a scale from 1 to 5 for clarity and easiness. All of them classified them as “very clear” (5). For easiness five users classified it as very easy (5), one user classified it as “easy” (4) and another one as “medium” (3).

6.3.2 Tool Use

The first question asked was about how easy is the tool to use. Six people said “very easy” and one said “easy”. In the Figure 6.5 is visible the response to the question about the usefulness of the alerts they have received.

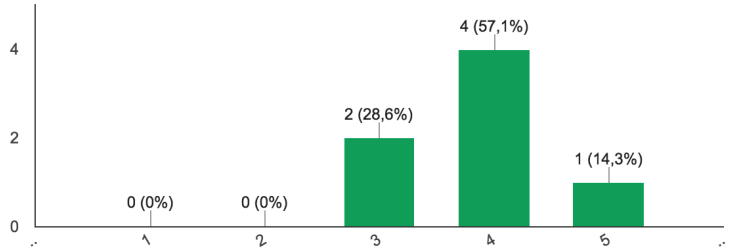

Figure 6.5: Users’ perceived utility of CodeAware notifications (0 - not useful, 5 - very useful)

Then it was asked about the usefulness of the types of probes that were already implemented in the prototype that they used. The summary is visible in Figure 6.6. Users were asked for suggestions of new types of probes. They could not come up with anything right away but one mentioned that the *DiffProbe* should not be triggered if it was himself changing the artifact.

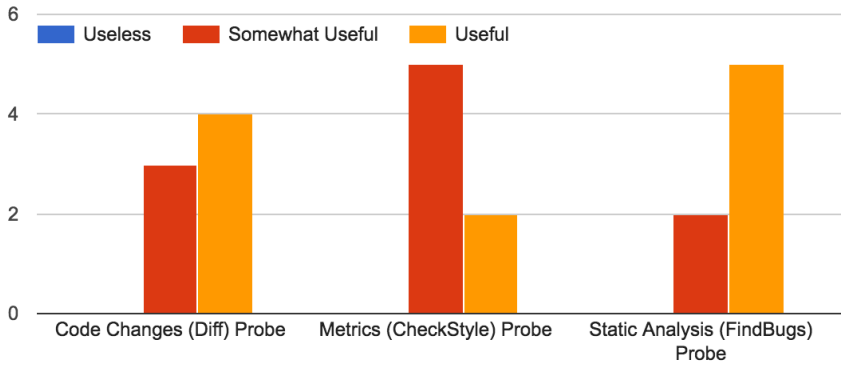

Figure 6.6: Users' perceived utility of the implemented probes

Suggestions for actuators were also asked. They suggested to have probes of type notification integrating with Slack[2], Skype[3] or even SMS (text messages) using Twilio[4]. All of the suggestions were very interesting.

When asked about other improvements that could be added to CodeAware, some users mentioned that there should be a way to hide the probes, this is, only show them to the owner because otherwise it can generate a no trust environment in the team. In addition, a user suggested that should be easier to add multiple probes at once.

6.3.3 CodeAware Use Cases

Users started to be asked whether CodeAware could help with code ownership, five said "yes", one said "sometimes" and another one said "unsure". One of CodeAware goals was to allow users to focus in specific parts of the project, five users agree that CodeAware helps the process and two users say that "sometimes" it helps. The same proportion of answers was reached when they were asked if CodeAware can help monitor and maintain code quality. Four users agree that CodeAware can improve team productivity while one says "sometimes" and two are "unsure".

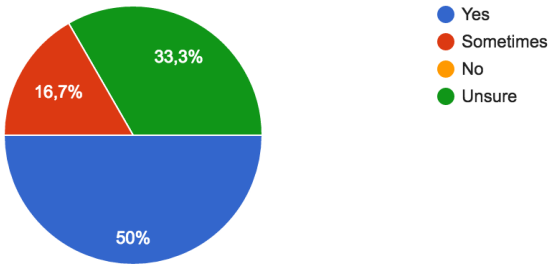

Figure 6.7: Results for the question "Could CodeAware amplify the benefits of Continuous Integration?"

[2] https://slack.com/
[3] https://www.skype.com/en/
[4] https://www.twilio.com/

Figure 6.7 shows the answer to the question "Could CodeAware amplify the benefits of Continuous Integration?". The results were not outstanding but it can be due to some users lack of experience with CI. Figure 6.8 shows whether users find beneficial CodeAware idea of developers being responsible for monitoring their part of the code over having a centralised approach. The intuition of the not so favourable answers to this question is related with the concept of shared code ownership. Users are afraid that the shared ownership can be broken by an individual keeping track of some parts of the code base.

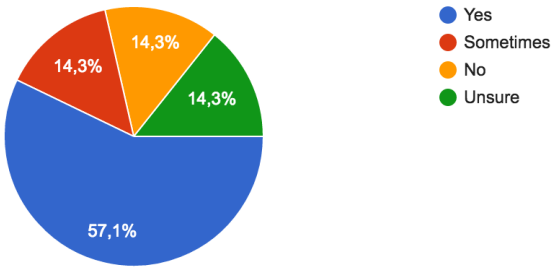

Figure 6.8: Results for the question "Do you think this approach would be beneficial over a centralized approach?"

Users were requested whether they would use CodeAware in its current form. Four people said "perhaps", two said "yes" and one did not answer. For a prototype, with a limited number of probes and actuators, the result to this answer is impressive. Figure 6.9 shows the answer to the question whether they would use CodeAware in a real team software projects if CodeAware became an extensible, professional-grade tool with a rich library of probes and actuators. From this answer it can be deducted that the enquired users liked and found interesting the general CodeAware concept.

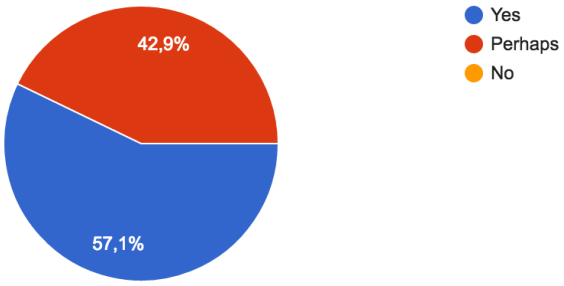

Figure 6.9: Results to the question "If CodeAware became an extensible, professional-grade tool with a rich library of probes and actuators, would you use it in a real team software project?"

In the end of the post-test users were asked to tell what they thought were CodeAware benefits and CodeAware shortcomings.

Citing users' perceived benefits:

- "Code Maintenance and Track".

- "The fact that a developer can focus in what he wants is a great advantage over a centralised approach. The IntelliJ integration is visually pleasant and very user friendly".
- "It can make me comfortable knowing when my code is changed. Making sure the change makes sense".
- "Checking if the code written by us was changed - better to maintain things working"
- "The probes are useful to find errors or conflicts".

Citing users' perceived shortcomings:

- "It is a bit costly to add probes mainly if we want to do it to many methods/classes. The fact that probes created are visible to everyone can generate a "bad environment" in a corporate environment, and in the current paradigm of "code ownership" in which everyone is responsible for the code.

When asked for suggestions, one suggested to have automatic probes however, he did not explain but perhaps he was thinking in meta-probes. And a user mentioned that the IntelliJ IDEA plugin could be improved to auto save probes and show more information in the panel rather than having to open a new window.

In general it can be said, by the answers given, that users understood the concept underlying CodeAware. Nevertheless, the kind of probe that they tend to prefer and talk more is the same they point more pitfalls, this is, the Diff probe. They like to know that their code is changed however, they are afraid that team code ownership can be ruined by it.

6.4 Conclusions

Users were very cooperative and provided some new insights. This validation was very important to understand what people think about the current state of CodeAware prototype and if they think that this vision has a future. The answers were in the direction that CodeAware can continue evolving.

Chapter 7

Conclusions and Future Work

CodeAware vision was initially narrowed down to fit this thesis scope. The main goal was to develop a prototype to prove CodeAware feasibility. The next steps included designing a modular architecture so that the system could be easily extensible. The development was the next phase. Nevertheless, along the development, architecture decisions kept being changed to better achieve the modularity goal. The external libraries and tools to use were also picked in this stage. During the development stage, there were weekly meetings with the CodeAware team (me, this thesis' supervisors and Uduak Eren) to define the development direction and decide features' priorities. In the end, a system validation was done with the collaboration of seven students from the last year of the Mestrado Integrado em Engenharia Informática e Computação from Faculdade de Engenharia da Universidade do Porto.

CodeAware concept was proven to be feasible with the prototype that was developed. Furthermore, it was achieved a very modular architecture so it can keep being extended providing a richer library of probes and actuators (Chapter 5 describes the easiness of this process).

In this chapter are presented the answers to the research questions, the challenges faced and the future work.

7.1 Answers to Research Questions

From the experiment done to validate CodeAware, were extracted some insights that help answer the research questions. The number of users tested is small but is representative of graduating software engineers as it is a sample extracted from the graduating students.

RQ1 *Do developers think CodeAware empowers team productivity?*

The initial hypothesis was that CodeAware could increase team productivity by allowing each developer to be focused in what he needs. Moreover, by encouraging developers to track what is happening in the code base should lead to a preventive behaviour.

The results from the experiment led to a similar conclusion. Four out of the seven users agreed that CodeAware can empower team productivity, one said "sometimes" and two said they are "unsure". Four of the users recognised that CodeAware when being used by a team can increase the productivity due to the monitoring tools that CodeAware provide. The results are good when it is realised that users used a prototype containing only three types of probes and two types of actuators. Or users are forecasting a promising future with more and more types of probes being added or they can be already sensing that what is provided is already able to help a team empower productivity.

Nevertheless, users pointed out that in an extreme situation the fact that probes are seen by everyone could drive to a no trust environment. In the sense that if a developer is attaching a probe to an artifact, it is because he does not trust others to deal with it. This can point the team productivity in the opposite direction. It is imperial that this caveat is further explored in order to understand how it can be addressed to avoid a negative impact in team productivity. This new insight proved that a system validation is very important because the CodeAware team have never thought about this situation.

RQ2 *Do developers think that CodeAware can help monitor and maintain code quality?*

By providing probes that can inspect the code and actuators that warn developers if the inspections return unwanted results, it is expected that CodeAware can immediately allow developers to know when code quality drops. CodeAware prototype already provides two types of probes that allow analysis of code quality. They are a static analysis using *FindBugs* and a metric analysis with *CheckStyle*.

The users in the experiment agreed that CodeAware could help in this aspect when assessing the quality of the probes' types provided. Five said that *FindBugs* is "useful" and two said it is "sometimes useful". For the metrics (*CheckStyle*) the result was the opposite of the *FindBugs*. This is due to the fact that metrics was not a so known concept between the tested users. Nevertheless, it is a good result.

From the comments provided during the experiment it is clear that most of the users were impressed with the static analysis probe provided by *FindBugs*. They liked the fact that it immediately pointed to the problem and its explanation was clear. The fact that they could instantly know if someone touched their code was also viewed by them with interest because they could assess if the changes made sense.

RQ3 *Do developers find CodeAware useful?*

The users that tested CodeAware found it useful, some of them (two out of seven) would use it in the current form, this is, as a prototype and five perhaps would use it. If the CodeAware probes and actuators library was richer the numbers increase. Four would use it and three perhaps. Nobody said that would not use CodeAware. It is important to note that we are aware that CodeAware needs to be easily extensible to easily achieve a rich library of probes and actuators.

Furthermore, three out of six users say that CodeAware amplifies the benefits of Continuous Integration, one says "sometimes" and two are "unsure". Four out of seven say CodeAware distributed approach is beneficial over the Continuous Integration centralised one, one says it is "sometimes", another was "unsure" and one does not think it has any benefits. This results are tricky because only three out of the seven users had already practised CI in at least a project. The ones that had already used CI recognised CodeAware benefits.

7.2 Contributions

CodeAware vision is completely new. There is nothing like this out there that we are aware of, what opens a big opportunity for CodeAware. CodeAware changes the centralised paradigm of CI allowing every developer participating in a project to specify their interest in artifacts.

The fact that CodeAware focus both in team productivity and code quality can increase the success of the projects. It increases the development speed and by monitoring the code base the code quality stays at high level. CodeAware can help a team manage the technical debt of a project.

From the system validation performed it is clear that CodeAware can have a bright future if it keeps being actively developed. CodeAware is already a consistent system but it still has a big margin to grow.

It is possible to imagine companies in the future relying in CodeAware because they are already using CI and CodeAware extends it. If it is proved that in fact CodeAware can empower team productivity then it will be a huge attraction to companies, which are always looking for methodologies to empower productivity. For that, CodeAware needs to get more mature and enrich its library. Later on, allowing a company to test CodeAware in a real project can be what is needed to step up CodeAware.

7.3 Limitations

CodeAware ecosystem for now is limited to the Java programming language. It is like this because it was needed one language to begin and Java is widely used. Notwithstanding, in the future other programming languages can be added.

CodeAware platform is limited to probes and actuators in the platform. A developer can extend the platform by developing new probes and actuators types. It is however not possible for a developer to create a new probe/ actuator type on the fly when attaching it to an artifact.

CodeAware is still a prototype whose scope was narrowed down to fit this master thesis. It should grow in the direction to include the cut out functionalities.

7.4 Challenges

CodeAware is the simple concept of moving some configurations of CI from a centralised approach to a distributed one. But underneath, what makes it work, requires a complex architecture. The first

challenge was to decide how to achieve a good architecture. It could be an architecture in multiple sequential steps or an architecture with steps running parallelly. Parallelly would certainly achieve a better performance but it brings another technical challenges. To overcome the problems it was decided to use Akka that allowed the achievement of this parallelism seamlessly.

From the beginning of the development was clear that CodeAware would have to be easily extensible. This was a key point that required a very careful design of the architecture. Fortunately, it was achieved.

The fact that the code is always evolving led to the problem of probes becoming orphan, for that it was added the *HostResolution* actor to the original architecture, in the IDE part a warning is also presented when a probe becomes orphan.

There are some challenges that still need to be overcame like grouping probes that do exact the same thing in order to have them run only once. This is, have an *Optimiser* actor between the *Dispatcher* and the *SignalGenerator*. For now, they just run multiple times.

Providing users with a user-friendly environment to manage CodeAware ecosystem (CodeAware IntelliJ IDEA plugin) required some brainstormings, but there is still work that has to be done in this area.

7.5 Future work

The CodeAware engine, this is, CodeAware Jenkins plugin achieved a good maturity. Currently, it is modular and completely functional. What needs to be done however, is adding an Optimiser as referred in the previous section (Section 7.4).

CodeAware IntelliJ IDEA plugin has a lot to improve to make developers life much easier. Functionalities to provide a smoother user experience should be added. For example:

- **mass copying/ modification of probes and actuators** — copy/ modify multiple probes at the same time.
- **amplify and dampen probes** — applying a probe to all methods of a certain type or to all classes in a package. And the other way around, delete all probes of certain type under certain conditions.
- **Improving reuse** — for now new probes and actuators can be cloned from existing ones in the project. Having a library with already set up ones would be beneficial.
- **Probes traceability** — When modifying a probe, that was cloned from another, it is intuitive that the developer may want to change the original one as well.

Probes ownership should be added as well to deal with the tested users' concerns related to probes privacy (refer to Chapter 6).

Finally, more probes and actuators have to be developed to enrich the CodeAware library. They are easy to be developed as explained in the Chapter 5.

References

[AEP15] Rui Abreu, Hakan Erdogmus, and Alexandre Perez. Codeaware: Sensor-based fine-grained monitoring and management of software artifacts. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 551–554. IEEE, 2015.

[BCG+10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[Bec98] K Beck. Extreme Programming: A Humanistic Discipline of Software Development. *Fundamental Approaches to Software Engineering*, page 1999, 1998.

[Bec99] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications (2nd Edition)*. Addison-Wesley Professional, 1994.

[CP13] G. Ann Campbell and Patroklos P. Papapetrou. *SonarQube in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2013.

[Cun92] Ward Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, December 1992.

[DMG07] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[FF06] Martin Fowler and Matthew Foemmel. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, page 122, 2006.

[Fow03] Martin Fowler. TechnicalDebt, http://martinfowler.com/bliki/TechnicalDebt.html. October 2003.

[GS09] Georgios Gousios and Diomidis Spinellis. Alitheia Core: An extensible software quality monitoring platform. *Proceedings - International Conference on Software Engineering*, pages 579–582, 2009.

[Jen16] Jenkins. Jenkins continuous integration server. https://github.com/jenkinsci/jenkins/blob/4b2d83b0e4568d97a831916760f2c8af56bf7abb/README.md, April 2016.

[Kan02] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[KNO12] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. Technical debt: from metaphor to theory and practice. *Ieee software*, (6):18–21, 2012.

[Let12a] Jean-Louis Letouzey. SQALE method definition document, version 1.0. http://www.sqale.org/, January 2012.

[Let12b] Jean Louis Letouzey. The SQALE method for evaluating technical debt. *2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings*, pages 31–36, 2012.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[MBNC14] Kıvanç Muşlu, Christian Bird, Nachiappan Nagappan, and Jacek Czerwonka. Transition from centralized to decentralized version control systems: A case study on reasons, barriers, and outcomes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 334–344, New York, NY, USA, 2014. ACM.

[Mey14] Michael Meyer. Continuous integration and its tools. *Software, IEEE*, 31(3):14–16, 2014.

[MMMW05] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume*, pages 77–80. Society Press, 2005.

[Son16] SonarSource. Sonarqube^{TM}, version 5.3. http://www.sonarqube.org/, January 2016.

[Sta08] John A Stankovic. Wireless sensor networks. *IEEE Computer*, 41(10):92–95, 2008.

[VEM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97–106. IEEE, 2002.